Compiler and Runtime Support for Hybrid Static/Dynamic Scheduling

Hans Vandierendonck
School of Electronics, Electrical Engineering and Computer Science
Queen's University Belfast, University Road, Belfast BT7 1NN, United Kingdom
Email: h.vandierendonck@qub.ac.uk

***Abstract*—One of the outstanding issues in parallel computing is the selection of task granularity. This work proposes a solution to the task granularity problem by lowering the overhead of the task scheduler and as such supporting very fine-grain tasks. Using a combination of static (compile-time) scheduling and dynamic (run-time) scheduling, we aim to make scheduling decisions as fast as with static scheduling while retaining the dynamic load-balancing properties of fully dynamic scheduling. We present an example application and discuss the requirements on the compiler and runtime system to realize hybrid static/dynamic scheduling.**

I. Motivation

Tuning task granularity in task parallel programs is key to optimizing the performance of parallel programs [1]. Task granularity refers to the amount of work performed by a task. Making tasks too coarse-grain reduces the parallelism of the program, while making tasks too fine-grain exposes overheads of the task scheduler. In either case, scalability is affected.

Prior research has reduced the scheduling overhead. Cilk recursively decomposes tasks [2]. Task decomposition can be very deep but the scheduler is designed to provide low overhead on over-decomposed code. Lazy binary splitting avoids task decomposition if it is unlikely that the exposed parallelism will be utilized [3].

This work investigates static scheduling as a means to further reduce the minimum task granularity supported by a programming language and its runtime system. Static scheduling can support finer-grain tasks as scheduling decisions are made prior to the execution. The schedule is, however, rigid and cannot be adjusted, e.g., for load balancing purposes.

II. Example

Figure 1 shows the speedup of the *mcf* benchmark on an Intel Xeon E5-2650 with 8 cores (no hyper-threading) using static scheduling and using Cilk's dynamic scheduler. The parallelism is expressed identically in both cases. The graph shows that the Cilk scheduler, which is an efficient dynamic scheduler, introduces net overhead, while static scheduling provides performance improvement.

III. Software Support

In this work, we aim to obtain the performance of statically scheduled fine-grain code without modifying the the parallel programming language. As such, the compiler and runtime system must collaborate to transform the parallel Cilk program to an appropriate scheduled static program.

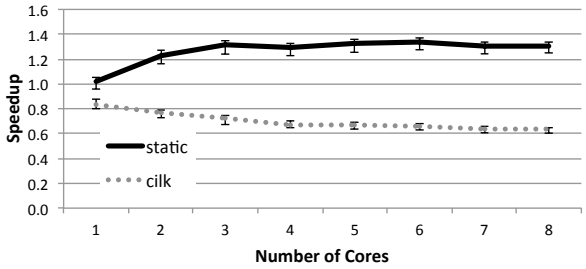

Fig. 1. Static versus dynamic scheduling on a loop in the SPEC CPU2006 *mcf* benchmark. The reference input is used.

The runtime system needs to schedule both conventional sequential tasks that are executed by a single thread and multi-threaded task bundles that are executed by multiple threads. The runtime systems needs to decide on the appropriate number of threads to execute for a multi-threaded task bundle, and needs to marshal those threads quickly enough.

The compiler transforms parallel code to statically scheduled multi-threaded tasks, or to parallel code using only sequential tasks. The translation involves the selection of an appropriate thread count. Alternatively, the compiler may generate multiple versions of the code with distinct thread counts. The runtime system may then decide what thread count to use for a multi-threaded task depending on the number of idle threads and on monitoring of past performance.

With such a system, we aim to reduce the granularity of parallel tasks that may be used in programs, thereby reducing the tuning of task granularity and ultimately making such tuning redundant. The resulting code will demonstrate increased performance and scalability.

Acknowledgment

This research is supported by the EPSRC under grant agreement no. EP/L027402/1 and by the European Union (FP7/2007-2013) under grant agreement no. 327744 (NovoSoft, Marie Curie Actions) and no. 619706 (ASAP).

References

[1] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson, "An experimental comparison of cache-oblivious and cache-conscious programs," in *SPAA*, 2007, pp. 93–104.

[2] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multi-threaded language," in *PLDI*, 1998, pp. 212–223.

[3] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin, "Lazy binary-splitting: a run-time adaptive work-stealing scheduler," in *PPoPP*, 2010, pp. 179–190.